Automatic Detection of Recursion Patterns

Jasper Van der Jeugt

Supervisor(s): prof dr. ir. Tom Schrijvers, Steven Keuchel

Abstract— **Rewriting explicitly recursive functions in terms of higher-order functions such as *fold* and *map* brings many advantages such as conciseness, improved readability, and it facilitates some optimisations. However, it is not always straightforward for a programmer to write functions in this style. We present an approach to automatically detect these higher-order functions, so the programmer can have his cake and eat it, too.**

Keywords— **catamorphisms, fold-build fusion, analysis, transformation**

I. Introduction

Higher-order functions are immensely popular in Haskell, whose Prelude alone offers a wide range of them (e.g., *map*, *filter*, *any*, ...). This is not surprising, because they are the key *abstraction* mechanism of functional programming languages. They enable capturing and reusing common patterns among, often recursive, function definitions to a much larger extent than first-order functions. In addition to the obvious code reuse and increased programmer productivity, uses of higher-order functions have many other potential advantages over conventional first-order definitions.

- Uses of higher-order functions can be more quickly understood because they reduce the that is already known pattern to a single name and thus draw the attention immediately to what is new (i.e., the function parameters).
- Because the code is more compact and the number of bugs is proportional to code size [1], higher-order functions should lead to fewer bugs.
- Properties can be established for the higher-order function independently from its particular uses. This makes (e.g., equational) reasoning more productive.
- Since properties and occurrences are more readily available, they make good targets for automatic optimisation in compilers.

A particularly ubiquitous pattern is that of folds or *catamorphisms*. In the case of lists, this pattern has been captured in the well-known *foldr* function.

$$\begin{aligned}&foldr :: (a \to b \to b) \to b \to [a] \to b\\ &foldr\ \_\ z\ [\,] \qquad\quad = z\\ &foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)\end{aligned}$$

Indeed, the research literature is full of applications, properties and optimisations based on folds [2, 3, 4, 5].

Hence, given all these advantages of folds, one would expect every programmer to diligently avoid explicit recursion where folds can do the job. Unfortunately, that is far from true in practice. For many reasons, programmers do not write their code in terms of explicit folds. This class comprises a large set of advanced functional programmers (see V-A). This is compounded by the fact that programmers often do not bother to define the equivalent of *foldr* for other inductive algebraic datatypes.

Yet, sadly, these first-order recursive functions are treated as second-class by compilers. They do not benefit from the same performance gains like loop fusion and deforestation. In fact, the leading Haskell compiler GHC won't even inline recursive functions.

We disagree with this injustice and argue that it is quite unnecessary to penalize programmers for using first-class recursion. In fact, we show that with a little effort, catamorphisms can be detected automatically by the compiler and automatically transformed into explicit invocations of folds for the purpose of automation.

II. Folds, Builds and Fusion over Lists

Catamorphisms are functions that *consume* an inductively defined datastructure by means of structural recursion. They can be expressed in terms of *foldr* [4].

The *build* function is the dual to the *foldr* function. Where the *foldr* function captures a pattern of list *consummation*, *build* captures a particular pattern of list *production*.

$$\begin{aligned}&build :: (\forall b.(a \to b \to b) \to b \to b) \to [a]\\ &build\ g = g\ (:)\ [\,]\end{aligned}$$

Given these functions, we can *fuse* the production and consummation of a list: meaning that no intermediate datastructure needs to be allocated. The foldr/build-fusion rule is given by:

$$foldr\ cons\ nil\ (build\ g) \quad \equiv \quad g\ cons\ nil$$

The main Haskell compiler, GHC, currently provides limited automation for foldr/build-fusion: by means of a GHC rewrite rule. When functions are defined explicitly in terms of *foldr* and *build*, after inlining, the rewrite rule may perform the fusion.

Unfortunately, this existing automation is severely limited. Firstly, it is restricted to lists and secondly it requires programmers to explicitly define their functions in terms of *build* and *foldr*.

This work lifts both limitations. It allows programmers to write their functions in explicitly recursive style and performs foldr/build-fusion for any directly inductive datastructure.

III. Discovering Folds and Builds

We have devised a set of formal rewrite rules to turn explicitly recursive functions into folds and to rewrite producers into builds where possible.

A. Syntax and Notation

To simplify the presentation, we do not use Haskell source syntax or even GHC's core syntax (based on System F). Instead, we use the untyped lambda-calculus extended with constructors and pattern matching, and (possibly recursive) bindings.

$$\begin{array}{llcl}
\text{binding} & b & ::= & x = e \\
\text{pattern} & p & ::= & K\ \overline{x} \\
\text{expression} & e & ::= & x \\
 & & \mid & e\ e \\
 & & \mid & \lambda x \rightarrow e \\
 & & \mid & K \\
 & & \mid & \mathbf{case\ e\ of}\ \overline{\mathbf{p \rightarrow e}}
\end{array}$$

The extension to GHC's full core syntax, including types, is relatively straightforward. We also need an advanced form of *(expression) context*:

$$\begin{array}{lcl}
E & ::= & x \\
 & \mid & E\ x \\
 & \mid & E\ \Box \\
 & \mid & E\ \triangle
\end{array}$$

A context E denotes a function applied to a number of arguments. The function itself and some of its arguments are given (as variables), while there are holes for the other arguments. In fact, there are two kinds of holes: boxes $\Box$ and triangles $\triangle$. The former is used for a sequence of unimportant arguments, while the latter marks an argument of significance.

The function $E[\overline{e}; e]$ turns a context E into an expression by filling in the holes with the given expressions.

$$\begin{array}{lcl}
x[\epsilon; e] & = & x \\
(E\ x)[\overline{e}; e] & = & E[\overline{e}; e]\, x \\
(E\ \Box)[\overline{e}, e_1; e] & = & E[\overline{e}; e]\, e_1 \\
(E\ \triangle)[\overline{e}; e] & = & E[\overline{e}; e]\, e
\end{array}$$

Note that this function is partial; it is undefined if the number of expressions $\overline{e}$ does not match the number of box holes.

B. Rules

The non-deterministic rules to rewrite functions into folds or builds can be found in Figure 1 and Figure 2 respectively. To keep the exposition simple, they are only given for lists and not arbitrary algebraic datastructures: the specifics for this generalization can be found in the full text. As an illustration, consider *map*, which is both a fold as well as a build.

$$\begin{array}{l}
map = \lambda f\ y \rightarrow \mathbf{case\ y\ of} \\
\quad [\,] \qquad\ \rightarrow [\,] \\
\quad (v : vs) \rightarrow f\ v : map\ f\ vs
\end{array}$$

Applying the rules for build detection yields us:

$$\begin{array}{l}
map = \lambda f \rightarrow \lambda l \rightarrow build\ (g\ f\ l) \\
g \quad\ \ = \lambda f \rightarrow \lambda l \rightarrow \lambda c \rightarrow \lambda n \rightarrow \\
\qquad \mathbf{case\ l\ of} \\
\qquad\quad [\,] \qquad\ \rightarrow n \\
\qquad\quad (y : ys) \rightarrow c\ (f\ y)\ (g\ f\ ys\ c\ n)
\end{array}$$

Now g can be recognise by our fold detection rules:

$$\begin{array}{l}
g = \lambda f \rightarrow \lambda l \rightarrow \lambda c \rightarrow \lambda n \rightarrow \\
\quad foldr\ (\lambda y\ ys \rightarrow c\ (f\ y)\ ys)\ n\ l
\end{array}$$

IV. Implementation

We implemented these transformations on top of GHC [6], the de-facto standard compiler for Haskell. Since GHC 7.2.1, a plugin framework is available [7].

This plugin framework facilitates writing GHC Core [8] transformations. Using this framework, we no longer have to edit the GHC source code in order to add, modify or delete Core-to-Core passes: we can do so in a plugin. This plugin is then passed to GHC using a command-line argument.

We implemented such a plugin, which contains passes:

- A pass to convert functions written in explicitly recursive style into functions in terms of folds. This pass is a deterministic implementation of the rules in Figure 1. Directly inductive datastructures other than list are also supported.
- We have a similar pass for builds: a deterministic implementation of the rules in Figure 2.
- A custom inliner, over which we have a bit more control than the default GHC inliner. However, we ultimately decided not to use this custom inliner, so it is disabled by default.
- An implementation of foldr/build-fusion for any algebraic datatype for which we have a fold and a build. By using this pass, we can avoid having to add extra rule pragmas [9].

Additionally our work also contains Template Haskell [10] routines to mitigate the burden of defining folds and builds for algebraic datatypes. These folds and builds can be generated by issuing, e.g., for a type *Tree*:

$ (*deriveFold* '*Tree* `"foldTree"`)
$ (*deriveBuild* '*Tree* `"buildTree"`)

V. Evaluation

A. Identifying folds

A first aspect we can evaluate is how well our detection of folds works. Unfortunately, manually identifying folds in projects very labour-intensive. This explains why it is especially hard to detect false negatives.

Additionally, very little other related work is done. There is the *HLint* [11] tool is able to recognize folds as well, but its focus lies on refactoring rather than optimisations. We cannot directly compare our results with those of HLint since we detect folds over all possible directly inductive datastructures, and HLint is limited to list. Hence, we classify the folds we detect as either a fold over a list or a fold over some other datastructure.

Table I shows that we clearly detect more folds than the HLint tool does. Additionally, we also tried our tool

$$\boxed{b \leadsto b'}$$

$$\text{(F-Bind)}\ \frac{\begin{array}{c} e_1' = [x \mapsto [\,]]e_1 \qquad f \notin fv(e_1') \\ E[\overline{u}; y] = f\ \overline{x}\ y\ \overline{z} \qquad ws\ fresh \\ e_2 \overset{E}{\leadsto}{}^{vs}_{ws}\ e_2' \qquad \{f, x, vs\} \cap fv(e_2') = \emptyset \end{array}}{\begin{array}{c} f = \lambda\overline{x}\ y\ \overline{z} \to \mathbf{case\ y\ of\ \{[\,] \to e_1; (v : vs) \to e_2\}} \\ \leadsto f = \lambda\overline{x}\ y\ \overline{z} \to foldr\ (\lambda v\ ws\ \overline{u} \to e_2')\ (\lambda\overline{u} \to e_1')\ y\ \overline{u} \end{array}}$$

$$\boxed{e\ {}_{x}\overset{E}{\leadsto}_{y}\ e'}$$

$$\text{(F-Rec)}\ \frac{e_i\ {}_{x}\overset{E}{\leadsto}_{y}\ e_i' \quad (\forall i)}{E[\overline{e}; x]\ {}_{x}\overset{E}{\leadsto}_{y}\ y\ \overline{e'}} \qquad \text{(F-Refl)}\ \frac{}{e\ {}_{x}\overset{E}{\leadsto}_{y}\ e}$$

$$\text{(F-Abs)}\ \frac{e\ {}_{x}\overset{E}{\leadsto}_{y}\ e'}{\lambda z \to e\ {}_{x}\overset{E}{\leadsto}_{y}\ \lambda z \to e'} \qquad \text{(F-App)}\ \frac{e_1\ {}_{x}\overset{E}{\leadsto}_{y}\ e_1' \qquad e_2\ {}_{x}\overset{E}{\leadsto}_{y}\ e_2'}{e_1\ e_2\ {}_{x}\overset{E}{\leadsto}_{y}\ e_1'\ e_2'}$$

$$\text{(F-Case)}\ \frac{e\ {}_{x}\overset{E}{\leadsto}_{y}\ e' \qquad e_i\ {}_{x}\overset{E}{\leadsto}_{y}\ e_i' \quad (\forall i)}{\mathbf{case\ e\ of\ \overline{p \to e}\ {}_{x}\overset{E}{\leadsto}_{y}\ case\ e'\ of\ \overline{p \to e'}}}$$

Fig. 1. Fold discovery rules

	Total	List	ADT	HLint
Cabal	20	11	9	9
containers	100	11	89	1
cpphs	5	2	3	1
darcs	66	65	8	6
ghc	327	216	111	26
hakyll	5	1	4	0
haskell-src-exts	37	11	26	2
hlint	6	3	3	0
hscolour	4	4	0	2
HTTP	6	6	0	3
pandoc	15	15	0	2
parsec	3	3	0	0
snap-core	4	3	1	0

TABLE I
RESULTS OF IDENTIFYING FOLDS IN SOME WELL-KNOWN PROJECTS

	Total	List	ADT
Cabal	101	81	20
containers	25	2	23
cpphs	6	5	1
darcs	354	354	0
ghc	480	178	302
hakyll	22	18	4
haskell-src-exts	140	74	66
hlint	69	62	7
hscolour	33	33	0
HTTP	11	11	0
pandoc	97	97	0
parsec	10	10	0
snap-core	4	4	0

TABLE II
RESULTS OF IDENTIFYING BUILDS IN SOME WELL-KNOWN PROJECTS

on the test cases included with HLint – which we could all correctly detect. This suggests that we detect a strict superset of possible folds, even for lists. The fact that the number of possible folds in these projects found by HLint is so low indicates that the authors of the respective packages might have used HLint during development.

B. Identifying builds

We also evaluated the detection of builds. However, as far as we know no work has been done in this area – hence, we cannot compare these results to anything meaningful. The results are shown in Table II. We conclude that they are within the same order of magnitude as the number of folds we found.

C. Foldr/build-fusion

Unfortunately it remains a hard problem to count how many times foldr/build-fusion can be applied in a Haskell package. The concrete difficulties we currently face are:

- Manual intervention is not required to do the detection of folds or builds, but in order to actually perform the transformation some manual changes are required: more precisely: adding imports and language pragmas, and using Template Haskell to derive the necessary folds.
- Code which is crucial to the performance of a package is often manually optimised. Hence, we usually find no opportunities to remove intermediate datastructures there.
- A lot of code is partially written using higher-order functions (e.g. from the *Data.List* library) and partially using

explicit recursion. Due to practical problems, we currently cannot perform any fusion there, although we are working to lift this limitation.

D. Benchmarks

We researched the speedups caused by foldr/build-fusion. In order to do this, we constructed small pipeline-like programs who have the inherent property of being fusable. Consider the functions:

```
sumt :: Tree Int → Int
sumt (Leaf x)       = x
sumt (Branch l r) = sumt l + sumt r
mapt :: (a → b) → Tree a → Tree b
mapt f (Leaf x)       = Leaf (f x)
mapt f (Branch l r) = Branch (mapt f l) (mapt f r)
uptot :: Int → Int → Tree Int
uptot lo hi
        | lo ⩾ hi      = Leaf lo
        | otherwise =
          Branch (uptot lo mid) (uptot (mid + 1) hi)
    where
      mid = (lo + hi) ‘div‘ 2
```

These auxiliary functions allow us to write functions which must allocate intermediate datastructures, as the result of *uptot* and *mapt*. Hence, we can benchmark the functions:

```
t1, t2, t3, t4, t5 :: Int → Int
t1 n = sumt (1 ‘uptot‘ n)
t2 n = sumt (mapt (+1) (1 ‘uptot‘ n))
t3 n = sumt (mapt (+1) (mapt (+1) (1 ‘uptot‘ n)))
t4 n = . . .
t5 n = . . .
```

$$\boxed{b \rightarrowtail b' ; b''} \qquad \text{(B-BIND)}\ \frac{c, n, g \text{ fresh} \qquad e \;{}_f\overset{c,n}{\rightarrowtail}_g\; e'}{f = \lambda\overline{x} \to e \;\rightarrowtail\; f = \lambda\overline{x} \to build\ (g\ \overline{x});\ g = \lambda\overline{x} \to \lambda c \to \lambda n \to e'}$$

$$\boxed{e \;{}_f\overset{c,n}{\rightarrowtail}_g\; e'} \qquad \text{(B-REC)}\ \frac{}{f\ \overline{e} \;{}_f\overset{c,n}{\rightarrowtail}_g\; g\ \overline{e}\ c\ n}$$

$$\text{(B-NIL)}\ \frac{}{[\,] \;{}_f\overset{c,n}{\rightarrowtail}_g\; n} \qquad \text{(B-CONS)}\ \frac{e_2 \;{}_f\overset{c,n}{\rightarrowtail}_g\; e_2'}{(e_1 : e_2) \;{}_f\overset{c,n}{\rightarrowtail}_g\; c\ e_1\ e_2'}$$

$$\text{(B-BUILD)}\ \frac{}{build\ e \;{}_f\overset{c,n}{\rightarrowtail}_g\; e\ c\ n}$$

$$\text{(B-CASE)}\ \frac{e_i \;{}_f\overset{c,n}{\rightarrowtail}_g\; e_i' \qquad (\forall i)}{\mathbf{case\ e\ of\ \overline{p \to e}} \;{}_{\mathbf{f}}\overset{\mathbf{c,n}}{\rightarrowtail}_{\mathbf{g}}\; \mathbf{case\ e\ of\ \overline{p \to e'}}}$$

Fig. 2. Build discovery rules

The results can be found in Figure 3. We can see that the speedups are very significant, even when fusion can only be applied once (i.e., in *t*1). We also see that the more we can apply foldr/build-fusion, the more significant our speedup becomes.

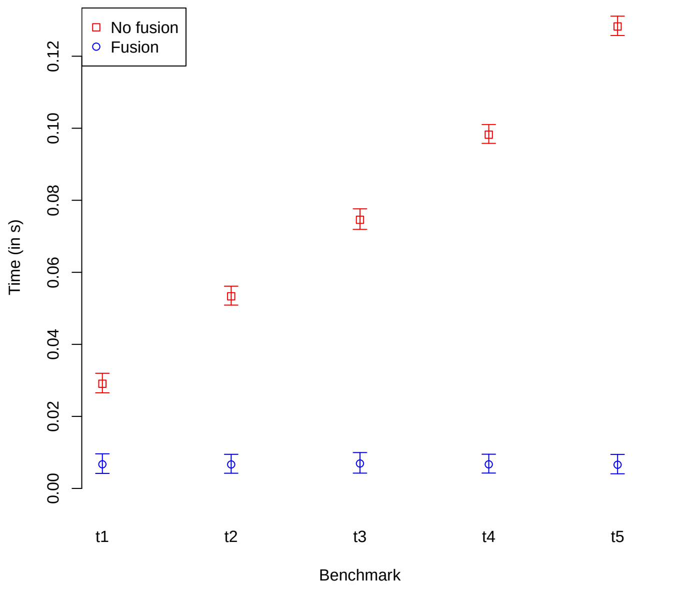

Fig. 3. Benchmarking the specified functions with and without fusion

VI. Related Work

A. Applications of Folds and Fusion

There is a long line of work on writing recursive functions in terms of structured recursion schemes, as well as proving fusion properties of these and exploiting them for deriving efficient programs.

Bird and Meertens [12, 13] have come up with several equational laws for recursion schemes to serve them in their work on program calculation.

With their Squiggol calculus, Meijer et al. [4] promote the use of structured recursion by means of recursion operators. These operators are defined in a datatype generic way and are equipped with a number of algebraic laws that enable equivalence-preserving program transformations.

Gibbons [14] promotes explicit programming in terms of folds and unfolds, which he calls *origami* programming. Unfolds are the dual of folds, and capture a special case of builds.

Gill et al. [3] present the foldr/build-fusion rule and discuss its benefits. They mention that it would be desirable, yet highly challenging, for the compiler to notice whether functions can be expressed in terms of *foldr* and *build*. That would allow programmers to write programs in whatever style they like.

Stream fusion [15] is an alternative to foldr/build-fusion. It has the benefits of easily being able to fuse zips and left folds. However, at the time of writing, there is no known reliable method of optimising uses of *concatMap*. *concatMap* is important because it represents the entire class of nested list computations, including list comprehensions [16].

B. Automatic Discovery

The *HLint* [11] tool is designed to recognize various code patterns and offer suggestions for improving them. In particular, it recognizes various forms of explicit recursion and suggests the use of appropriate higher-order functions like *map*, *filter* and *foldr* that capture these recursion patterns. As we already showed in Section V-A, we are able to detect more instances of folds for Haskell lists than HLint. Moreover, HLint makes no effort to detect folds for other algebraic datatypes.

Sittampalam and de Moor [17] present a semi-automatic approach to *foldr* fusion based on the MAG system. In their approach, the programmer specifies the initial program, a specification of the target program and suitable rewrite rules. Then the MAG system will attempt to derive the target implementation by applying the rewrite rules.

VII. Conclusions

Programmers prefer high-level programming languages in order to be able to write elegant, maintainable code. However, performance remains important, i.e. the code needs to be able to be translated to efficient assembly. The conflict between those two goals can be solved fairly well by using advanced compilers and smart optimisations.

We added such a smart optimisation. It allows the programmer to write small, composable functions, either explicitly recursive or using a higher-order fold. More complex functions van be expressed as a combination of these small building blocks. When compiled naively, this would impose penalties on the performance, because of the overhead of allocating, building and destroying intermediate datastructures. By using foldr/build-fusion, we can avoid this overhead.

Our thesis has lifted the limitation that functions have to be written in a certain style in order to benefit from foldr/build-fusion, and extends it to work on any directly inductive datastructure instead of just lists.

Future work. A number of routes are still open for exploration. In particular, we mention mutually recursive functions, mutually recursive datatypes and GADTs [18]. Folds and builds exists for these types [19, 20], but our detection rules are incapable of finding these.

References

[1] John E. Gaffney, "Estimating the number of faults in code," *Software Engineering, IEEE Transactions on*, vol. SE-10, no. 4, pp. 459–464, 1984.

[2] Philip Wadler, "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, no. 2, pp. 231 – 248, 1990.

[3] Andrew Gill, John Launchbury, and Simon L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the conference on Functional programming languages and computer architecture*, New York, NY, USA, 1993, FPCA '93, pp. 223–232, ACM.

[4] Erik Meijer, Maarten Fokkinga, and Ross Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," 1991, pp. 124–144, Springer-Verlag.

[5] Graham Hutton, "A tutorial on the universality and expressiveness of fold," *J. Funct. Program.*, vol. 9, no. 4, pp. 355–372, July 1999.

[6] The GHC Team, "The glasgow haskell compiler," 1989.

[7] The GHC Team, "Extending and using ghc as a library: Compiler plugins," 2011.

[8] Andrew Tolmach and Tim Chevalier, "An external representation for the ghc core language," 2009.

[9] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare, "Playing by the rules: rewriting as a practical optimisation technique in ghc," in *Haskell Workshop*, 2001, vol. 1, pp. 203–233.

[10] Tim Sheard and Simon Peyton Jones, "Template meta-programming for haskell," *SIGPLAN Not.*, vol. 37, no. 12, pp. 60–75, Dec. 2002.

[11] Neil Mitchell, *HLint Manual*, 2006.

[12] Richard Bird, "Constructive functional programming," in *Marktoberdorf International Summer school on Constructive Methods in Computer Science*, M. Broy, Ed. 1989, NATO Advanced Science Institute Series, Springer Verlag.

[13] Lambert Meertens, "Algorithmics – towards programming as a mathematical activity," in *CWI symposium on Mathematics and Computer Science*. 1986, pp. 289–334, North-Holland.

[14] Jeremy Gibbons, "Origami programming," in *The Fun of Programming*, Jeremy Gibbons and Oege de Moor, Eds., Cornerstones in Computing, pp. 41–60. Palgrave, 2003.

[15] Duncan Coutts, Roman Leshchinskiy, and Don Stewart, "Stream fusion. from lists to streams to nothing at all," in *ICFP'07*, 2007.

[16] Duncan Coutts, "Stream fusion: practical shortcut fusion for co-inductive sequence types," 2010.

[17] Ganesh Sittampalam and Oege de Moor, "Mechanising fusion," in *The Fun of Programming*, Jeremy Gibbons and Oege de Moor, Eds., Cornerstones in Computing, pp. 79–129. Palgrave, 2003.

[18] James Cheney and Ralf Hinze, "First-class phantom types," 2003.

[19] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring, "Generic programming with fixed points for mutually recursive datatypes," in *ACM Sigplan Notices*. ACM, 2009, vol. 44, pp. 233–244.

[20] Patricia Johann and Neil Ghani, "Foundations for structured programming with gadts," in *ACM SIGPLAN Notices*. ACM, 2008, vol. 43, pp. 297–308.